Q

IN THE UNITED STATES DISTRICT COURT
FOR THE DISTRICT OF DELAWARE

SRI INTERNATIONAL, INC., a California Corporation, Plaintiff, v. INTERNET SECURITY SYSTEMS, INC., a Delaware corporation, INTERNET SECURITY SYSTEMS, INC., a Georgia corporation, and SYMANTEC CORPORATION, a Delaware corporation, Defendants.	Case No. 04-1199-SLR

Expert Report of Stuart Staniford

Table of Contents

Table of Figures

I have agreed to testify as an expert witness in this lawsuit, and am being compensated by ISS as detailed below. I expect to testify at trial concerning the matters discussed in this report if so asked by the court or by the parties' attorneys.

I) Background and Qualifications

I was educated initially as a theoretical physicist, first in England, and then obtaining a PhD in Physics from the University of California at Davis in 1993. I found myself most passionate about computer work, and decided to obtain a Masters of Science in Computer Science. I joined the computer intrusion detection group at UC Davis in 1994. This was the group that invented network intrusion detection in the late 1980s. I began in that group as a graduate student, and left in 1997 as an assistant adjunct professor.

While at UC Davis I initially worked on methods to track hackers back to their true locations. Then I led a team of other researchers and students that developed GrIDS, the first intrusion detection system aimed at wide area networks that used a multi-layer hierarchy. Next I was asked by DARPA (the Defense Advanced Research Projects Agency) to lead the development of the Common Intrusion Detection Framework (CIDF) – an effort to get all the intrusion detection research systems funded by DARPA to interoperate together.

In 1997, I founded my own company Silicon Defense, with the intention of doing further DARPA research. During that time, I became co-chair of an Internet Engineering Task Force working group that developed an intended standard for intrusion detection research (IDWG). I then went on in recent years to work primarily on the problem of computer worm spread, and the quantitative performance of algorithms for preventing worm spread. I currently make my living as an independent consultant working primarily in these areas.

My research papers on computer intrusion detection, worm spread, and worm containment have been cited by other researchers over 1200 times according to scholar.google.com in March 2006.

For full details, see my curriculum vitae attached as Appendix A.

II) Opinions to be Expressed, and the Bases and Reasons for Those Opinions

I understand that the claims at issue are:

- **all claims** of patent #6,321,338 (the **338** patent)
- **claims 1-6 and 12-17** of patent #6,484,203 (the **203** patent)
- **claims 1-6 and 13-18** of patent #6,711,615 (the **615** patent)
- **all claims** of patent #6,708,212 (the **212** patent).

I understand that a claim is *anticipated* by prior art when one prior art reference discloses all elements of the claim. I understand that a claim is *obvious* when all elements of that claim are disclosed in two or more prior art references and there was a motivation to combine those references. Where I have combined references, there was an explicit suggestion in one of the references regarding the combination, a technique was or discussed so widely in prior art that anyone skilled in the art would have known of its availability for implementation.

I understand that all references dated on or before November 8th, 1997 (the *statutory bar date*) are prior art. I discuss my opinion in detail below.

At trial, I may provide a tutorial on the technical background relating to computer network security. This may include a discussion of the following key concepts in computing, networking, statistics, and security.

IIa) Definition of Common Computer and Network Terms

A modern computer consists of a series of hardware components required in order to support the software that the user wishes to run. The *cpu* is a piece of hardware (nowadays a single *integrated circuit*, or *chip*), which interprets low level software instructions and carries them out. These instructions are typically at the level of "add A to B", or "find the contents of memory location X and bring them into the chip". Millions of these instructions together make up a modern software program.

The instructions that make up a program are referred to by software engineers as *code*. The actual low-level code that the processor operates with is known more specifically as *object code*. Object code is inconvenient for programmers to work with, since it consists of an enormous string of inscrutable numbers from the perspective of a human. Thus a variety of *computer languages* have been designed which allow programmers to express programs in a more human-friendly (or at least programmer-friendly) manner. This human-friendly code is called *source code*. Special programs called *compilers* and *linkers* translate the source code into object code that the computer's cpu can process efficiently. A large variety of computer languages are available for various purposes. Some popular ones mentioned in this report are C, Perl, and Prolog. The term *application* is synonymous with program (though occasionally application refers to what is a suite of distinct programs from the perspective of the computer).

In addition to one or more cpus, a computer requires *random access memory* (RAM, frequently referred to informally as just *memory*). Memory consists of specialized integrated circuits used to store software programs and their data in a form that can be rapidly accessed by the cpu. The amount of memory available to computers has increased from only a few kilobytes – thousands of bytes - in early years to gigabytes - billions of bytes - today. A byte consists of eight individual ones or zeros and is enough to store a single character of text.

Programs that are actually running and working will be kept in memory so that the cpu can read the object code as needed to execute the instructions. In addition to its code, the program will usually have areas set aside for the data it is working with (eg the word processor I am using to write this report will have allocated memory space for the data that makes up my document as I type it in). An instance of a program and its data in memory is known as a *process*.

Numbers are also often stored in memory, and depending on the number of bytes allocated for the storage of a number, a smaller or larger number can be stored. This concept will become important to us later, so let me go into just a little detail. A number that uses only a single byte can take only one of 256 values – 0 to 255. This is known as a *1-byte quantity*. However, if we use two bytes, then we can store numbers from 0 to 65535 in a *2-byte quantity*. A *4-byte quantity* can range from 0 to 4294967295 (a little more than 4 billion different possible values could be stored in the 4-byte slot in the memory). So, in short, in a small space of memory, we can only store small numbers, but as we add more bytes the size of the number that can be stored goes up exponentially.

Any computer requires a special suite of software known as the *operating system*. The operating system is responsible for organizing the operation of all other software programs on the computer, and providing those programs with access to the services of the hardware in a convenient and controllable way. For example, the operating system is responsible for deciding which programs will be able to use how much memory, working with the memory hardware to ensure that programs cannot alter each other's memory, and so forth. Also, the operating system is responsible for deciding which processes should run on the cpu at any given time. In modern computers, which are *time-sharing*, the operating system will rapidly swap between all the programs that would like to run at any given time. This provides the illusion (to humans) that all the programs are running at the same time. Typically, an operating system provides a set of *system calls* which are special operations that other programs may request the operating system to perform – this is the way that the operating system provides its services.

Random access memory is relatively expensive, and has the property that all the data in it is lost when the power is turned off. For this reason, cheap persistent ways to store data have been developed. The dominant technology at present is *magnetic hard drives*. These consist of spinning disks covered with a material that can be magnetized in several different directions. One direction represents zero, and another direction represents one, and by this means digital information can be stored and read by special heads that move over the spinning surface of the disk. Contemporary hard drives allow for years-long storage of very large amounts of data (up to terabytes - trillions of bytes), but are much slower to access than RAM.

Data on disks is not stored in a completely random way, or it would be impossible to keep it organized and critical data would get lost or accidentally overwritten. Instead, the operating system keeps data on disks organized into *files*. Each file gets one or more parts of the disk to store its data in, and the operating system ensures that different files get different parts of the disk. Special files called *directories* (or *folders*) keep lists of other files, and allow the user to view and organize their files in a tree-like hierarchy. When a user creates a new file, *eg*. a word processing document, the word processing software will communicate with the operating system, which will then allocate space on the disk for the new file, and then add data to it on request. A variety of system calls are concerned with operating system provision of these services – typically there are system calls to create a new file, to open an existing file to read or write to it, to actually perform a read or write of some data, together with calls to manage and traverse directories.

More sophisticated operating systems maintain an *audit trail* of actions that occur on the computer system. These can be very detailed – every time a file is opened for reading or writing, a new program is started or stopped, a user logs in or logs out, or the system interacts with the network, the actions are logged in the audit trail. This allows a very detailed historical record of who did what to what to be maintained. Audit trails are very voluminous and, depending on how much data is recorded, may have a significant performance impact on the computer. In some cases, every system call from a program to the operating system can be recorded in the audit trail. As we shall see, a number of early computer intrusion detection systems centrally depended on these audit trails.

Computers communicate with one another via *computer networks*. Physically, networks consist of wires, wireless access points which communicate via radio signals, and a variety of interconnection devices known as *hubs*, *switches*, *routers*, etc. However, computer scientists have defined a variety of abstractions and protocols that allow any computer to communicate with any other via computer networks which almost all interoperate to form the Internet, together with various private organizational networks which connect to the Internet. The physical details of the hardware are abstracted away.

At the lowest level of abstraction, computers exchange *packets*, which are sequences of bytes grouped and transmitted together. On modern networks, packets range in size from a few tens of bytes to about 1500 bytes.

The Internet is a large collection of networks of many different kinds all connected to each other. The network presents the impression of a single fabric in which any computer can connect to any other, and it does this by agreeing on a set of rules for how data is formatted into packets and how the packets are transmitted across the network. These rules are known as *network protocols*. Multiple protocols are involved in making the Internet work, and I need to explain several of them for understanding the issues in this case.

The Internet protocols are designed with multiple protocols that do different jobs. The aim of this structure is to keep individual protocols simple enough that they can be designed, implemented, and tested with manageable effort, and yet the whole structure of all the protocols together presents an enormous amount of functionality. The way protocols are combined together is called *layering*.

The Internet has a design with four *protocol layers*. At the lowest level is the *physical layer protocol* associated with the particular kind of network hardware. Above that are, in order, the *network*, *transport*, and *application layers*. We will take these in turn, and explains what it means for them to be layered over each other.

We start with the physical layer. There are a large number of different physical layer protocols. However, the most common is the *Ethernet* suite of protocols. This protocol, in its simplest form, formats packets as follows:

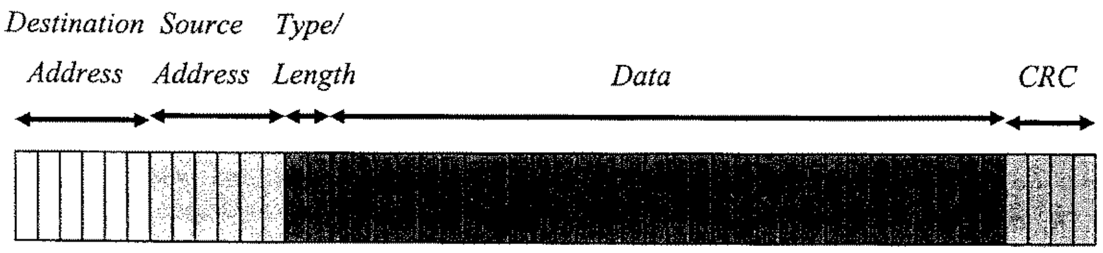


Figure 1: Format of an Ethernet packet

The way to read this kind of diagram is left to right. Each little box represents one byte (eight bits each encoding a one or zero). The bytes are to be transmitted one at a time over the network. The different colored regions represent different parts of the message that are interpreted for particular purposes. The first region of six pale blue bytes on the left of the diagram is the *Ethernet destination address* (essentially a long number which denotes a particular network interface on a particular computer). All types of Ethernet hardware recognize Ethernet addresses, and every computer Ethernet interface is assigned a unique address by its manufacturer. This guarantees that two computers will never have an Ethernet address in common.

Ethernet began as a Local Area Network (LAN) protocol. Originally, the LAN would consist of a few to a few hundred computers, likely in a single building. The medium was a broadcast medium, where every computer would essentially share the same network. Any packet would be sent to every computer on the LAN, and only the one with the destination address would read it.

This feature of Ethernet was particularly attractive in the early days of network intrusion detection as we will see later: a computer on the network could set its network interface into a special *promiscuous mode* where it would listen to **all** packets to and from all computers on the network (and try to detect intrusions into them).

Nowadays, Ethernet switch devices have become more sophisticated about not broadcasting packets to computers that will not need them, and the networks have become much larger in some cases, but the protocol is still a locally oriented protocol.

The next six-byte field (green) consists of the source address of the computer that generated the packet. This allows for return communication. If A begins a conversation with B, B will know where to send the reply because A's source address is in this second field in the packet. Next comes a two-byte field (red), which, depending on the exact version of Ethernet in use is either an explicit length of the data in the packet, or is a type code indicating the higher level protocol carried by the packet. Next comes the purple region of data bytes. This is data that is *opaque* as far as the Ethernet protocol and network hardware is concerned. Ethernet does not interpret this data in any way, but simply passes it along to the destination, which must interpret it. It is inside here that higher-level protocols occur, but Ethernet is not aware of their details. Finally, at the end of the packet is a CRC checksum (a technical means to ensure that if the packet is somehow corrupted in transit, that fact can be detected).

Many protocols share with Ethernet the feature that there is a *header*, which is involved in deciding where the packet should go and other details associated with transmitting it efficiently and correctly. And then there is the *body* or *data*, which is the actual data that is supposed to be transmitted.

Because Ethernet, like other physical layer protocols, is primarily local in its orientation, it cannot serve as a basis for a global network like the Internet. The Internet conceptually is a "network of networks". It uses a standard set of protocols (abbreviated to TCP/IP after two of the most important ones) to allow many different physical networks to all interoperate with each other and work together to present the illusion to Internet users that all the computers in the world are connected with each other.

An overview of the situation is shown in this next picture.

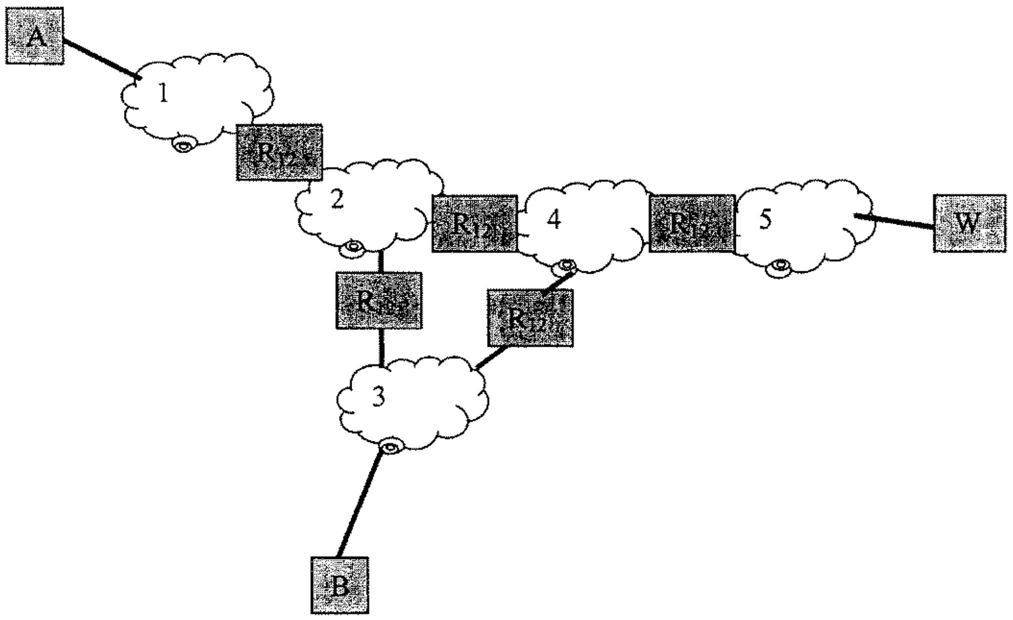


Figure 2: Illustration of networks and routers combining to form the Internet

Here, the clouds represent different physical networks, while the tan square boxes represent computers connected to the networks. A and B are user computers (desktops and laptops), while W is a web server. In the network context, computers at the edge of the network are frequently referred to synonymously as *hosts*, or *machines*. The blue boxes are *routers* – special devices that transition packets from one physical network to a different one. (Of course the real Internet involves millions of networks and hundreds of millions of end computers – this picture shows just a few to illustrate the idea).

When A wishes to see a web page, its packets requesting the web page must travel via networks 1, 2, 4, and then 5. Packets from W containing the requested web page must reverse this path, and travel through 5, 4, 2, and 1. Requests from B take a path through 3, 4, and 5, and the responses go through 5, 4, and 3.

Now the physical networks labeled 1, 2, 3, 4, and 5 may use quite different kinds of technology and cannot communicate with each directly, instead interacting via devices called routers (on which more in a moment). A number of protocols are involved in making all these disparate networks talk to each other in such a way that packets can go from anywhere in the network to anywhere else.

The most important of these is the *Internet Protocol (IP)*. This is the packet format that almost all data traveling across the Internet uses. IP has its own set of addresses – *IP addresses* -- that are distinct from the physical layer addresses and can be used for routing information long haul over the Internet. The general idea is that IP packets are encapsulated inside physical layer packets and routed from one network to another. A packet traveling from host A in our diagram above to host W will be created as an IP packet at A, and then wrapped in a physical layer packet and placed onto network 1 with the physical layer address of router R_{12}, but the IP address of web server W. When the network delivers this packet to R_{12}, the router will discard the physical layer header, look at the Internet address in the Internet packet, decide that the packet needs to go via network 2 and Router R_{12}, and then add a new and different physical layer header to the packet and send it on its way across network 2. This process repeats until the packet arrives at its destination.

Thus the routers are critical to allowing many different physical networks to collaborate to form the Internet. A packet going from a home computer in California to a web server in New York might cross a wireless Ethernet in the user's home, a DSL line to an Internet Service Provider (ISP), a wired Ethernet at the ISP, several fiber optic networks of the large carrier ISPs that provide the backbone networks that span the country, before going to the (likely) Ethernet network of whoever owns the web-server. Thus the same IP packet has gone through what is often 10 to 20 different physical networks of a variety of different types. On each of those networks, it has been encapsulated with the right kind of physical layer packet and sent on to the physical destination layer address of the next router in the hop. All the while, the IP packet has continued to carry the same destination address (that of W).

The format of the IP packet header is shown below. Fields that are not important to this report are not described for the sake of brevity and clarity (and are gray in Figure 3).

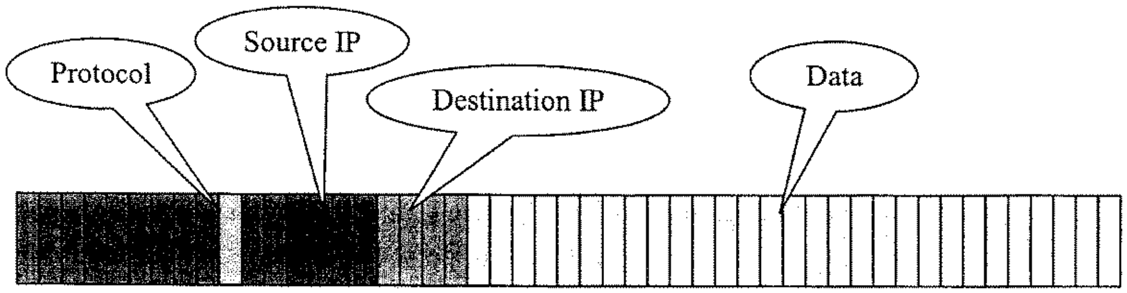


Figure 3: Format of an IP packet (selected fields only).

The green byte is used to denote the protocol used for the data portion (at right). As with Ethernet, IP has an opaque data section used to carry data for higher-level protocols and applications. However, the protocol byte is used to denote which protocol governs the data in the data section. This allows the operating system on the receiving computer to decide what software should be run on the data section.

The other things of importance to us in the header are the addresses. There are two four-byte IP addresses, one each for the source (the original computer that originated the IP packet), and for the destination (the end computer that will receive the packet). These are known as the *source IP address*, or simply *source IP* for short, and *destination IP address*, or, correspondingly, the *destination IP*. It is this destination address that the routers use to decide which neighboring router to send the packet to in its next hop, and via which physical network.

The IP protocol is defined to be a *best effort* delivery service. This means that the protocol effectively asserts that it will try to deliver the packet, but does not guarantee that it will succeed. If the packet is lost in transit for some reason, IP will not do anything to recover the situation. Packets can be lost for a variety of reasons: most commonly, congestion at a router (too many packets) can cause the router memory for packets to fill up, and then incoming packets must be abandoned because there is no space to store them until the congestion problem is alleviated.

Additionally, the protocol does not guarantee to deliver the packets in the order they were sent. Packets can arrive out of order, for example, if there are multiple routes between the source and destination and some of the packets go one way and some the other. (IP is also stateless – it doesn't remember anything about past packets it has delivered when deciding what to do with the current packet, so if it has a choice of routes, it isn't guaranteed to send all the packets between a particular source and destination via the same route)[1].

From the perspective of an application, this isn't an entirely satisfactory situation. If a web server is sending a web page to a user's computer, and the page is big enough to be

[1] This is in contrast, for example, to the traditional phone network, which is a circuit-switched network – when a call is placed a circuit is set up and devoted to that particular call. The circuit delivers all data in order.

split across a number of packets, the user expects to see the web page with all the elements in the right order, and without missing pieces. Thus most applications need any missing data to be retransmitted, and the data *reassembled* to ensure it is in the right order.

To avoid every single application having to develop separate computer code to solve these problems, a standard protocol, *transmission control protocol (TCP)*, has been developed. TCP is one instance of what is known as a *transport protocol*; protocols that provide additional transport services over and above just best-effort network delivery protocols such as IP. It is TCP that lends its name to the common shorthand *TCP/IP*, which denotes the entire suite of protocols that make the Internet work.

TCP is an extremely complex protocol that has evolved significantly over time. I will only present a few of the simpler aspects of it here. More details can be found in texts such as [Comer]. I reserve the right to go into more detail as additional issues arise in the course of the lawsuit.

The basic goal of TCP is to provide a two-way transmission channel between an application on one computer that wishes to talk to another application on a different computer. The application that initiates the conversation is called the *client* and the application on the computer receiving the request to talk is called the *server*. TCP provides a (potentially) long-lived channel by which these two applications can talk across the network, and the protocol guarantees that all data sent in each direction will be delivered in order, and if it should go missing, TCP will retransmit it. This communication channel is known as a *TCP connection*, sometimes just *connection*, or *network connection*, for short. The "connection" terminology can be used of other protocols, but with no other qualifier usually implies TCP, and always has a connotation of a persistent channel that can carry data that will not fit in a single packet.

TCP connections can transmit data in both directions. The two data streams "slide" past each other without interfering in any way in transit. We speak of the *forward stream*, to refer to the data from the client to the server, and the *reverse stream* for the data coming back from the server to the client. Each stream is separately ordered.

TCP is used by our web-server above, for example. The web-browser on a user computer will act as a TCP client and initiate a TCP connection with the server and send a request over the forward stream. The web server will respond with the data for the web page over the reverse stream.

So how is this functionality achieved?

TCP breaks up the streams into pieces that will fit into a packet. It takes the data that the application would like to transmit, and makes a *TCP segment* by adding a header to it. That segment is fit into an IP packet and then transmitted over the network via the IP protocol. The format of a TCP segment is as shown below. However, as with IP, I will not describe all fields in the header, but rather only the ones I need to explain at present as background for my analysis later in the report. The rest are grayed out.

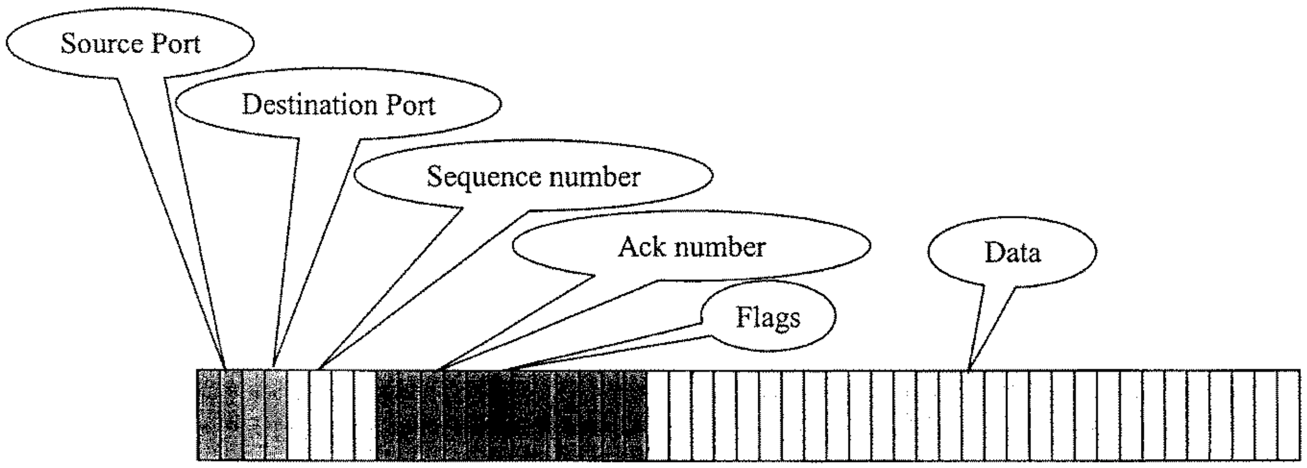


Figure 4: Format of a TCP segment (selected fields only).

TCP does not carry addresses to distinguish different computers. That is taken care of by the IP packets that the TCP packets are wrapped in. However, since a computer can run many applications at the same time, it is necessary for the portions of the operating system that handle interaction with the network protocols to decide which application is supposed to get the data from a particular connection.

To this end, TCP has an abstraction known as a *port number*. Conceptually a computer on the network presents a number of numbered abstract *TCP ports* to which another application on a different computer could potentially *connect* (ie form a connection). Each of these ports is potentially connected to a different application. Both the client software that initiated the connection and the server software that accepts it (or not) have a port number. These port numbers are carried in the TCP header as shown above. The source port is the port number connected to the process on the machine that sent the packet, and the destination port number is associated with the process that will receive the packet when it reaches its destination.

The operating system code responsible for networking is known as the *network stack*. When an application is waiting for a packet, it issues a system call to the operating system and then waits. When the network stack receives a packet, it checks the destination port number, looks up the appropriate waiting process, and then causes the system call to return with the application data from the TCP segment. If the application is present but not waiting, the network stack will simply *buffer* (store in memory) the data until the application is ready.

There are certain *well-known ports* that are standardized to mean particular applications. Thus for example, the world-wide-web uses a protocol called *HTTP* to transport web pages, and the servers for HTTP that serve web pages almost always *listen* on port 80 (also known as the *HTTP port*). Meanwhile email is sent over a protocol called *SMTP*, which listens on port 25. Computer scientists often speak of the *network services*, or just *services*, that computers offer on different ports (such as HTTP). Thus service is usually equivalent to destination port.[2] Thousands of ports have special meanings for well-known services, but since a two-byte port number can take on values from 0 to 65535, there are still plenty of ports that the operating system can use for temporary purposes.

The purpose of the source port is to ensure that the stack knows how to send a response packet back to the application at the other end. This is often (but not always) a non-standard port that the network stack assigns temporarily at random.

The process of setting up TCP connections will turn out to be important, so we describe it here. TCP uses *sequence numbers* (the four-byte yellow quantity in Figure 4) to keep the data in order. These sequence numbers start with a random value (to make it hard for hackers to guess them and insert spurious data into TCP connections). However, the two ends of a connection must at the outset of a connection agree on sequence numbers so that they can both keep straight where they are up to later.

The way this is accomplished at the outset of the connection is with some special packets that perform a *three-way handshake* between the two endpoints. The packets are distinguished by the flags field (the purple byte in Figure 4), which contains a series of bits that, if they are one instead of zero, give the packet special meaning. The next figure illustrates the handshake:

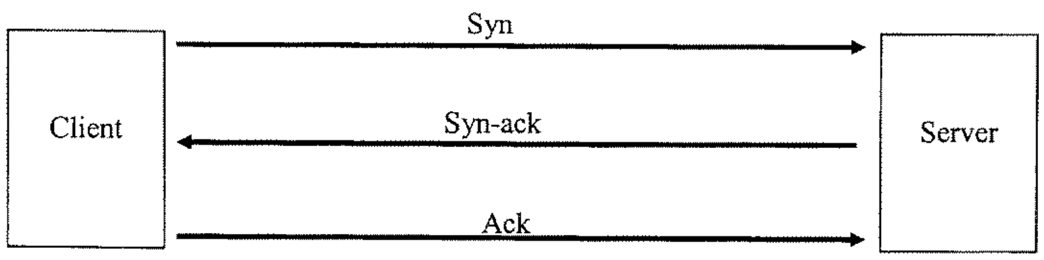


Figure 5: Exchange of packets in a TCP handshake

The handshake begins with the client initiating a *connection request*, by sending a *syn packet*, which is distinguished by having the *syn-bit* set in the flags. (When a bit is *set* it means that it is one rather than zero). This also sends the server the *initial sequence number* that the client proposes to use in the forward data stream.

The network stack on the server checks that there is an application on that port waiting for new connections, and that this source is allowed to connect to it. If these checks are passed, then the stack will create a packet known as a *syn-ack*, in which both the syn and *ack-bit* are set in the flags field. This packet also carries an acknowledgement of the forward stream initial sequence number (which is placed in the ack number field – the four byte rose colored field in Figure 4). Additionally, the server uses the sequence number field to propose an initial sequence number for the reverse stream. This TCP segment is wrapped in an IP packet and sent back over the network to the client.

[2] But not invariably. Occasionally, services are run on non-standard ports.

The last part of the handshake is for the client to send a TCP segment with just the ack-bit set. This confirms the reverse stream initial sequence number (in the ack field).

After the handshake is done, the applications can begin to send real data over the connection, which is now considered to be *set-up*. Usually, the conversation begins with a request from the client to the service on the server of some kind (but not always).

If something goes wrong when the server evaluates the initial request (*i.e.* the syn packet from the client), then the server will not respond with a syn-ack, but instead with a *reset packet*. This has the *reset bit* set in the flags field, and indicates that the network connection request has been denied.

To summarize, the protocol situation, the protocols are often shown stacked as follows:

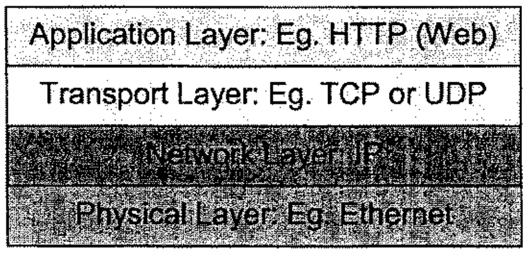

Application Layer: Eg. HTTP (Web)
Transport Layer: Eg. TCP or UDP
Network Layer: IP
Physical Layer: Eg. Ethernet

Figure 6: Layers in the Internet protocol stack

While these are conceived of being above one another (and the upper layers are called higher level protocols) in actual packets on the wire, the data of one protocol is inside the data of another. The headers occur in sequence, with the lowest level protocol header coming first, like this:

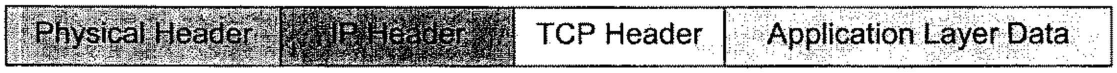


Figure 7: Protocol data layers as laid out in transmission on the network.

I now briefly describe some important application protocols that will be mentioned later in this report. These protocols run over TCP (i.e. they are *connection oriented)*, meaning that their data will be divided into segments that are packaged inside a TCP segment for transport.

Telnet is a protocol to support interactive login to a command line interface on a computer. It is an Internet standard [RFC 854]. Typically it runs over a single TCP connection with commands from the client to the server being sent in one direction, and responses from the server coming back in the other direction. A telnet session typically begins with a login and password being supplied, but then telnet itself does not impose much structure on the content of the connection. Telnet was historically a clear-text (unencrypted protocol), and is becoming less common for that reason, but was the most widespread application for interactive computing in the early to mid 1990s. The client to server communication was typically human-typed text, though not necessarily so.

FTP (File Transfer Protocol) is an Internet standard for moving files from a server to a client. It was very widely used in the early and mid1990s, but has been partially displaced by HTTP in the last decade. FTP makes use of a *control connection*, over which the client gives a series of standardized FTP commands (historically mainly via interactive human typing, but it can also be automated). The commands allow for such operations as listing directories, changing directories, and uploading or downloading files. FTP can be used to access private directories after supplying a username and password, but it was also commonly used as a way to make repositories of files available to the public via *Anonymous FTP*, in which anyone could log in as the user *Anonymous* and access a set of files and directories for which public permissions were granted.

SMTP (Simple Mail Transfer Protocol) is the main protocol used for sending of electronic mail over the Internet. It, like telnet, is a single connection protocol, but it involves a variety of standardized commands by which mail servers exchange mail with each other and move mail towards the recipients.[3]

HTTP (HyperText Transfer Protocol) is the main protocol supporting the World Wide Web, and it has become the dominant application protocol on the Internet. It's main function is to allow web browsers (as clients) to request web pages from servers. However, it has been extended to support all manner of features. It is a request-response protocol in which a request from the server is encoded in an HTTP header, and then a response comes back with the requested page. Various *HTTP methods* are similar to commands in that they support either getting data, or posting data to the server, or other operations. Different versions of the protocol and the client and server software may use one or multiple connections for exchanging requests and responses.

UDP (User Datagram Protocol). Although TCP is much the most common transport protocol in the TCP/IP protocol stack, it is not the only one. The second most important transport protocol is UDP, which is simpler than TCP. Essentially, it provides the same port number concept as TCP, meaning that it is possible for two applications on different computers to identify each other and carry on a remote conversation. As with TCP, there are well-known UDP ports for various network services. However, UDP does not provide reliable in-order delivery. Instead, it simply wraps one UDP packet in one IP packet and sends it. Transport is then on a best effort basis. We now discuss a couple of applications that often run over UDP instead of over TCP.

NFS (Network File System) is a protocol that is used for file sharing over networks. The concept is that a set of files and directories on a *file server* across the network is *mounted*

[3] Personal computers use several protocols for downloading mail from mail servers, such as POP, IMAP, and even viewing messages via the web and HTTP. However, outbound mail is almost always carried by SMTP, and that is the main way mail servers exchange mail with each other.

on a local client. The operating system of the client computer then maintains the illusion that the files are on a local disk. In fact, when a user or application tries to open or read or write to a remote file, NFS packets are exchanged with the relevant file data so that the remote version is read or written as appropriate. In the early to mid 1990s, NFS predominantly ran over UDP (though more recent versions have increasingly relied on TCP). It has historically been the main file sharing mechanism for Unix and Linux computers.

SNMP (Simple Network Management Protocol) is a protocol originally designed for configuration and monitoring of networking equipment (such as routers and switches, but also many other components). SNMP has a language for defining *MIBS (Management Information Bases)* which are structured collections of data that define the configuration of a device, and or records of its operation. The protocol itself typically runs over UDP and includes various operations to get data from a MIB, set the values of data in a MIB, or receive asynchronous notification of events that the managed device deems important. SNMP is a widely used standard, and many devices host MIBS covering their operation. Special *network management software* is used to visualize and manage the devices via SNMP.

The TCP/IP suite of protocols is used for the public Internet. However, it is also used for almost all internal networks of companies and other organizations. These internal networks are typically referred to as *enterprise networks*. Enterprise networks, at least for large distributed enterprises, look very similar technologically to the Internet in the sense that they consist of a set of physical networks of possibly multiple types connected by routers. However, they are smaller than the Internet, and typically connect to the Internet in only a few well-controlled places. In the early years of the Internet, most networks were wide-open to the public network. Thus all computers that were connected via some company TCP/IP network could connect freely to any other computer at any other company, or at universities and to the computers private citizens connected via an ISP. There really wasn't any networking distinction between a company's TCP/IP network and the Internet; the enterprise network was just the portion of the Internet where the network hardware happened to belong to that particular enterprise.

With the rise of computer attacks in the 1980s (discussed later), it became clear that running important company functions over a network that was open to the Internet wasn't going to work. Thus *firewalls* were developed. A firewall, also known as a *gateway*,[4] is a special network element charged with managing the connection between two networks where some traffic is allowed to flow, but there are also many restrictions to prevent access to proprietary applications and computers by unauthorized personnel.

[4] The term gateway has been used in computer science in two overlapping ways. Prior to the mid 1990s, the term was most often used to be synonymous with router. More recently, it is more often been used synonymously with firewall. Since routers sometimes, but not always, function as firewalls, the distinction may or may not be important, and has to be determined from context.

Logically, the situation looks something like this.

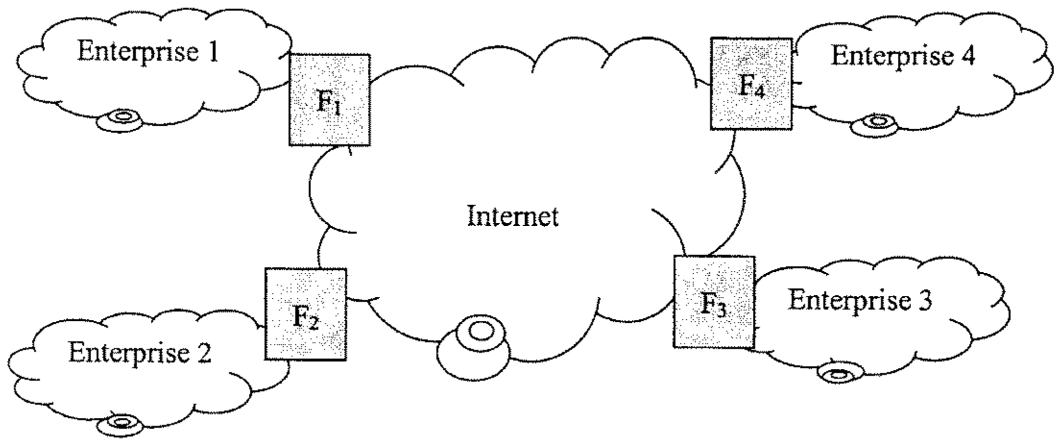


Figure 8: Logical view of enterprise networks connecting to the public Internet

Here, each of the enterprise networks are connected to the public Internet via one of the firewalls (F_1 – F_4). The firewalls prevent staff at Enterprise 1 casually trying to make connections to, say, accounting computers at Enterprise 2. This is of course a simplification – many organizations have more than one firewall connection to the Internet. Modern routers frequently have firewall functionality built into them, but standalone firewalls deployed in line with the routers are common also. Additionally, firewalls are used internally within enterprise networks to separate particularly sensitive parts of the network (eg the payroll!) from the bulk of employees.

An enterprise of any size is likely to have multiple offices in different cities. Of course, enterprises like to provide a convenient network environment where all the offices can share the enterprise network (eg access web servers, special applications etc) without noticing the geographical locations of network resources. Originally, enterprise networks were built by combining offices using networks over leased dedicated telephone lines. The internal routers on the enterprise network would control how packets moved between offices over the leased lines.

However, the trend since the mid 1990s has instead been to make enterprise networks into *virtual private networks (VPNs).* The next diagram illustrates the idea.

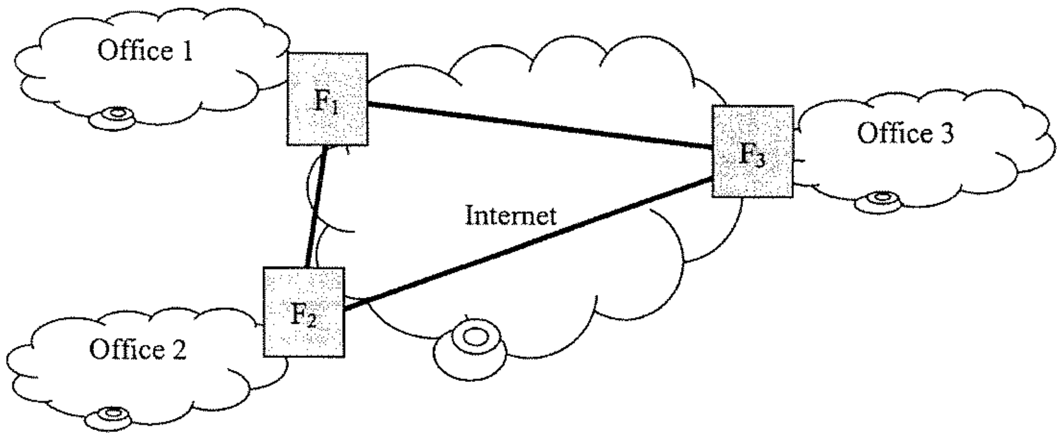


Figure 9: A Virtual Private Network as the enterprise network between multiple offices

The idea in a VPN is that the offices of an enterprise will be connected by special encrypted *tunnels* between the gateways with which each office is connected to the Internet. Packets going from office 2 to office 3 will first be routed to the firewall F_2, which will encrypt them, and then send them via the Internet to F_3, which will decrypt them and then route them on to the final destination in office 3. Many connections and other traffic between many hosts in each office may be transmitted over the same tunnel. The encryption makes it so that no routers or computers on the Internet can read the traffic (this is the *private* in "virtual private network"). However, from the perspective of computers in the various offices, the links between the various gateways don't look any different from any other network link in the enterprise network. The computers in the offices have the impression of a single enterprise network that works seamlessly. This is the *virtual* in "virtual private network".

As we discussed Figure 2 showing packets being routed across the Internet, or an enterprise network, a question might have occurred to the reader: how do routers know where to send things? The problem would seem to be a little complex – a router might see a packet destined for any address on the network, and in all cases it has to know what is the best route to send the packet forward on (or at least a reasonable choice that is not going to end with the packet wandering all over the network before reaching it's destination, or worse, going around in circles).

This is the job of a *routing protocol*. There are a number of important routing protocols associated with the TCP/IP protocol suite. We will first discuss some general issues, and then describe a particular routing protocol: the Open Shortest Path First (OSPF) protocol [RFC 2328, Comer Chapter 6]. The reason for this choice is that OSPF will be important to our discussion of certain intrusion detection systems later. OSPF is the most common routing protocol used for large enterprise networks.

Every router has a *routing table*. The routing table is a set of data in the router's memory that it uses to look up where to send a packet. Given an address, a router finds the appropriate entry in the table; in a simplified logical view, the entry will have the following three elements:

Destination network	Forwarding interface	Next router physical address

Table 1: Simplified logical view of a routing table entry (table row).

Basically, the router has to find the *destination network* that contains the destination address of the packet it is processing (I describe this further in a moment). Once it has found this and got the appropriate row in the routing table, it then knows that the IP packet needs to be sent out of the physical interface specified in the *Forwarding interface* column, and that the destination physical address of the router that will be the next hop in the packets journey is in the last column of the row[5].

In the way IP works, no router controls the entire path to the destination. Instead it just needs to answer the question of how to get the packet one hop closer to the destination. Then it trusts that next router to move the packet forward another hop – eventually the packet will get there.

So the routing problem is reduced to the question of how do the routers populate their routing tables with these entries? Clearly, if the routing table entries always move packets closer to their destination all will be well. However, inaccuracies in routing table entries could cause packets to go in the wrong direction (in the worst case circularly).

Now, one way that routing might have been done is that every address on the network could have its own entry in the routing tables. However, it should be clear this would pose scalability problems. If every router in the world (millions of them) had information about how to get to each individual computer in the world (hundreds of millions of them), then any time any computer changed its address or a new computer was added to the network, or an old removed, every router in the world would need to hear about it. The network would spend all its time doing nothing but propagating routing updates.

Instead, routing table entries are for whole *IP networks*, rather than individual entries. A network in the IP sense consists of a group of addresses that are all together. In fact, the way it is done is based on the bits in the address. If you recall Figure 3, an IP address contains four bytes, and since a byte has eight bits in, the address has 32 bits in. These bits are split, and some of them are used to denote the network address, while others are used to denote different hosts within the network. We can visualize the situation as follows (noting that each box in what follows is one bit, not an eight-bit byte as in earlier diagrams):

[5] This is a simplification – multiple routes and load balancing are beyond the scope of this document.

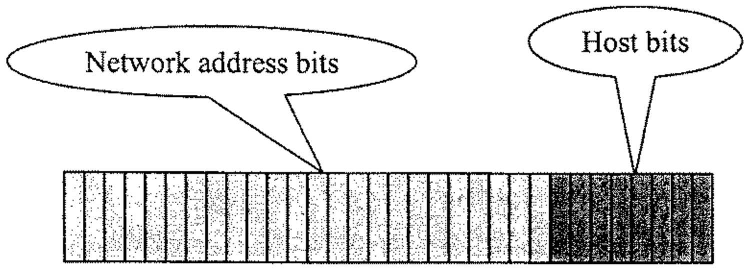


Figure 10: Division between network and host bits in an IP address

In this example, the first 24 bits are considered to be the network address, while the last eight bits are the host bits. In other words if two IP addresses have all the same bits in the first (green) 24 bits, they are considered to be on the same IP network. The last eight (blue) host bits distinguish different computers on the same network from one another.

Now, routing table entries are specified in terms of networks, rather than individual addresses. This allows larger numbers of computers to be grouped together for routing purposes. Only the router right next to them needs to distinguish them individually – the rest of the world only needs to know that the route to all hosts on that network goes to that router.

It is not mandatory that the split between the network bits and the host bits occurs after 24 bits. In modern versions of the IP protocol, this split is flexible and is part of the definition of the network.

However, even though addresses can be grouped like this, on a large enterprise network there can still be hundreds or even thousands of individual IP networks that the routers need to keep track of. How can all the routers learn where all the networks are and how to get to them? On small networks, the administrators can configure the route tables on all the routers by hand, but on large networks that change with any frequency at all (and all organizations change) this becomes completely impractical. This is the job of OSPF.

The concept of operations for OSPF is as follows. Every router maintains in its memory, in addition to the routing table as described above, a map of the entire network (the enterprise network in which OSPF is being run, that is). This map shows all routers within the network, as well as how they connect to each other (roughly equivalent to the entities in Figure 2, except not the individual computers on the edge). All routers have the same map[6] of the network, except during a transitional period after some kind of change in the network. This map is known as the *link-state database*. Whenever the map changes, the routers will recompute their routing table so that they know the best next hop for each packet that they might need to route.

[6] OSPF areas are beyond the scope of this document at present. This discussion treats the enterprise network as a single area (area 0).

To understand the way OSPF works, we need to understand both the way changes in the network are detected, and then the way in which all the routers on the network get to hear of the changes. The protocol relies on exchanging five different types of OSPF packets, namely:

- Hello packets,
- Database Description packets,
- Link State Request packets,
- Link State Update packets,
- Link State Acknowledgement Packets.

We will briefly describe all of these.

The way that routers hear about changes in the network is via a special OSPF packet called a *link state update (LSU)*. An LSU is transmitted inside an IP packet[7] from one router to an adjacent one (ie one separated by only a single network). The LSU structure is shown in *Figure 11*. It consists of a header, followed by a count, followed by a series of Link State Advertisements. The count tells the receiving OSPF software how many LSAs to expect.

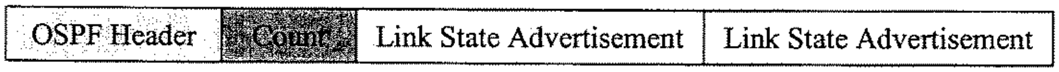


Figure 11: OSPF Link State Update Structure

The Link State Advertisements are a series of data elements that essentially tell the network that some aspect of the link database has changed. Link State Advertisements are flooded through the entire network, and when a router receives a set of LSAs, it checks to see if there are any that it is not already aware of, and if so it updates its link state database. Thus the LSAs are essentially a way of transferring the data in the link-state database from one router to another (so that they all have the same map of the network and don't get confused).

The format of the data being transferred in the LSAs is twenty bytes as follows:

[7] OSPF is unusual in being an application protocol that runs directly over IP instead of running over either UDP or TCP.

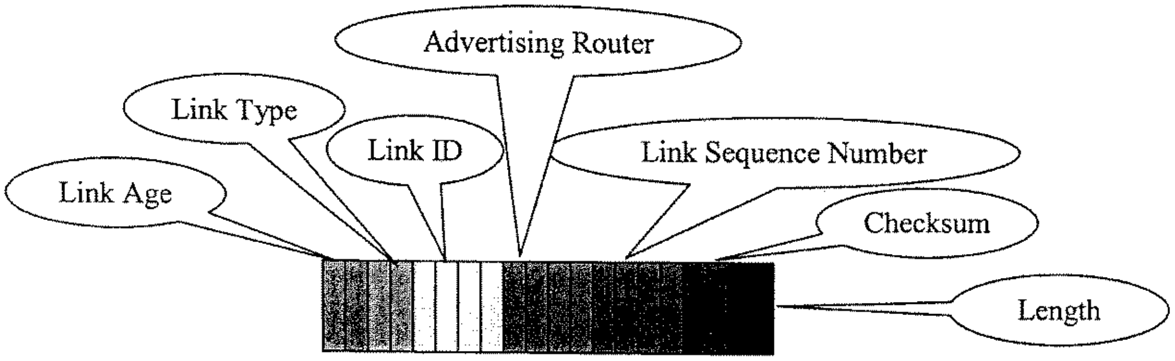


Figure 12: OSPF Link State Advertisement Structure

The flooding of LSAs wrapped inside LSU packets needs to be reliable – if an LSU gets dropped in the network, that fact needs to be detected and the LSU retransmitted. Otherwise, there would be a risk that no all portions of the network would hear about changes and the network would end up in an inconsistent state, which in turn might lead to routing errors. For this reason, when a router receives an LSU, it responds to the source with a Link State Acknowledgement packet. If the router that sent the LSU doesn't receive that acknowledgement after a fixed period of time, it will retransmit it.

When OSPF first begins, it does not know anything about the rest of the network. It needs to first find other neighboring routers who will tell it about the network. It does this by sending special Hello packets on the networks attached to all its interfaces (typically to a special address which all routers listen to, but the details vary depending on the type of physical network). Any neighboring routers present will respond in kind with another Hello packet. Once up and running, the routers continue to send Hello packets every ten seconds to detect if the networks and neighboring routers are working correctly. If Hello packets go unresponded to, the router concludes that a neighbor has disappeared, and it updates its link state database accordingly and then begins flooding LSAs to tell everyone else about the newly missing link in the network.

Once a new router has found a neighbor, or when two existing routers are linked for the first time, they need to synchronize link-state databases. They do this by exchanging a series of Database Description packets with its neighbors. These packets basically describe what links each router knows about and how old their knowledge is. Each router makes note of links where its neighbor has newer information than it does. It then issues Link State Request packets to get that newer information. The neighbor should respond with the information in Link State Update packets. By this means, the new router gets to the point where it has a link-state database and can thus compute a routing table to use in routing packets.

IIb) Background on Statistics

The patents in suit describe a statistical anomaly detection algorithm. In this section, I provide some background on statistics.

As my running example for illustration purposes, I am going to choose the following problem, which illustrates the statistical anomaly detection problem in miniature[8]. Suppose we have an enterprise network, and on the network is a server which runs some application that is important to the enterprise, and which is heavily used by employees. Let us further suppose that before using the application it is necessary to connect to it via a web browser (which will exchange data via HTTP over one or more TCP connections), and log in by giving a username and a password.

Most of the time, an employee who logs-in to the application will type his or her username and password correctly on the first attempt. However, sometimes a user will make a typing error without realizing and the login will fail. In that case, the user will try a second time, most likely with special care. It is less likely, but certainly within the bounds of normality, that a user will make another error and need to try a third time, or perhaps even more.

Furthermore, a user might forget their legitimate password, and make a number of guesses in an attempt to remember it. Eventually, such a user will either remember the password, or give up and get a new password some other way. However, it is occasionally possible that a user might make a few tens of login attempts by this means.

Attackers wishing to gain improper access might try to login to this application by guessing the password of some legitimate user – password guessing is one of the oldest hacking techniques, but it still works sometimes. Sometimes users choose weak, easily guessable, passwords (such as a child's name or their phone number). But, additionally, in the past, attackers have sometimes tried to guess passwords like this by *brute force*. That means they write a program which will attempt to login over the network over and over again, in each case trying a different password drawn from some dictionary of possible passwords. In this case, there might be hundreds of thousands of login attempts from the same source host.

Clearly, if we keep track of how many times a given source attempts to login before giving up for at least, say, five minutes, the password guessing hacker is going to look very unusual compared to the behavior of normal users of the application. The brute force attack will be a *statistical anomaly*. Let us try to explore this more carefully, and think about ways that such a statistical anomaly might be detected automatically in order to enhance the security of the network.

If we were to make a graph that shows on the x-axis the time of day at which a user attempts to login, and on the y-axis the number of login attempts before they either succeed or give-up, it might look as follows, for some particular hour from 8am to 9am at the beginning of some workday:

[8] The main background on computer security and intrusion detection will come later in this document, but I chose a very simple security application here.

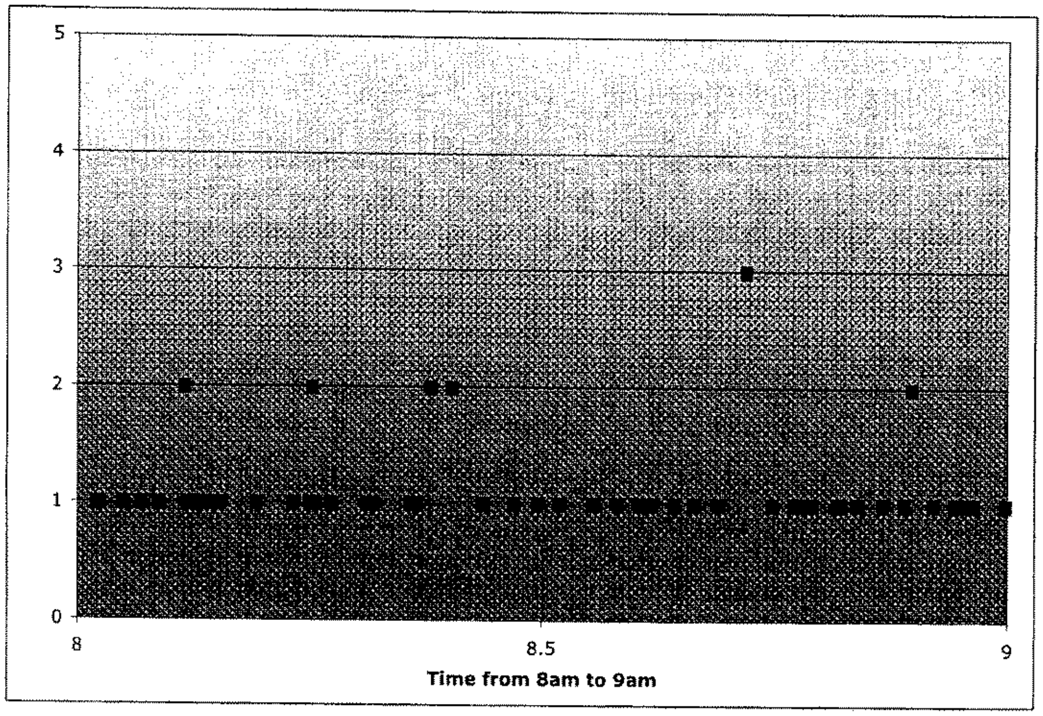


Figure 13: Illustration of hypothetical login attempt count sequence. Data is synthetic for illustration purposes only

The idea here is that each square represents an occasion when someone attempted to login. The height of the square (on the y-axis) is the number of attempts before they succeeded, and the horizontal (x-axis) position is the time at which that user began to login. Clearly, in this hour, we mostly saw users succeeding on the first attempt (1 on the y-axis), but occasionally it took 2 or 3. During this hour, no-one needed more than three attempts, but this is only a single hour. If we gathered statistics for a longer period of time, we would probably see occasional cases where users took more than three attempts to login. Still, if we were to see a user make 200 attempts, we might say that was unusual, relative to the history. However, looking at a graph of every instances like this would quickly become unwieldy if we wanted to look at a whole week, or a month.

A way to capture a longer-term picture is to make a histogram of everything that happens over some longer period of time. Suppose we were to take a day's worth of login data, count all the times users succeeded on the first attempt, all the times it took them two attempts, and so on. Then we divided these counts by the total number of all logins to get the percentage. That might look like this:

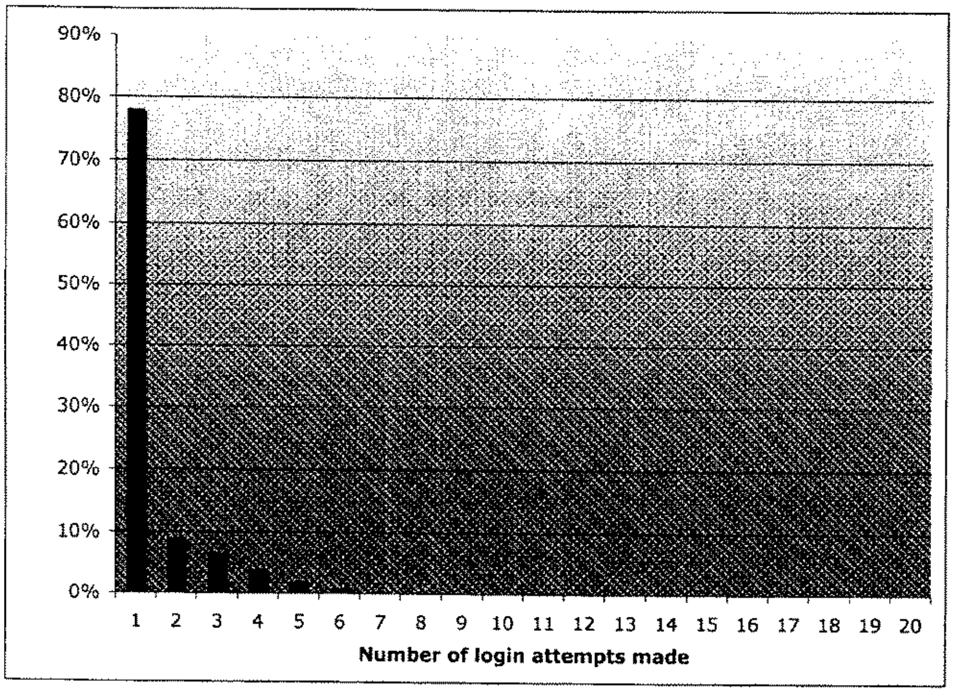


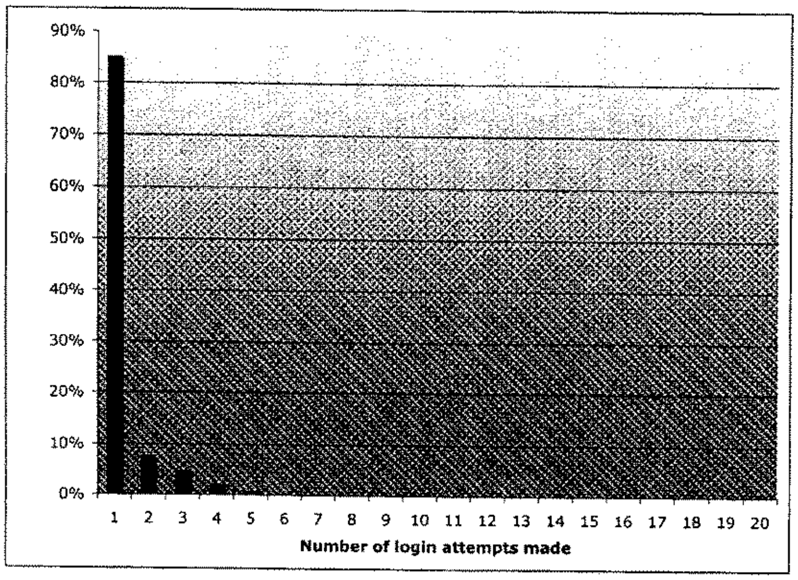
Figure 14: Histogram of occurrences of number of attempts on login in some hypothetical day of login attempts to an enterprise application

Here, the first column expresses the fact that in 78% of logins, users succeed on the first occasion. The second column says that in 9% of cases, users require exactly two attempts. And so forth. The columns tail off towards the higher values, and thus we refer to the sides where the values start to get small as the *tail* of the histogram. Obviously, the case where a user makes 200 attempts without succeeding in logging in is way off the right hand side of the histogram. It is far into the tail, and this intuitively tells us that it is a *statistical anomaly*, though we will make that concept more precise later.

The height of the bars here is an approximation to the *probability* that a particular situation will occur. The probability that something will occur is essentially a technical formalization of what is popularly known as "the odds" that something will happen. If, over a very large population of the something, some condition will hold X% of the time, then we say the probability of that occurring is X%. The *probability distribution* of something is essentially a list of all the cases that might occur, and the odds that each of them will happen. All the probabilities must add to 100% (if not, we haven't successfully enumerated all the possibilities in the situation, or we have miscalculated some of the probabilities). As we shall see, probability distributions are at the heart of statistical profiling (though with a lot of wrinkles to handle various practical issues of concern).

A histogram is essentially a way of visualizing the probability distribution of something – we lay out the possible cases on the x-axis, and then show the probabilities as the heights of the bars.

However, if we look at the histogram for a different day, we probably will not get quite the same values as on the first day. The degree or typing error of the application users is subject to chance, and therefore the proportions will not always be identical. If we re-run the experiment, we might get this histogram (with the first one shown just below it for comparison).



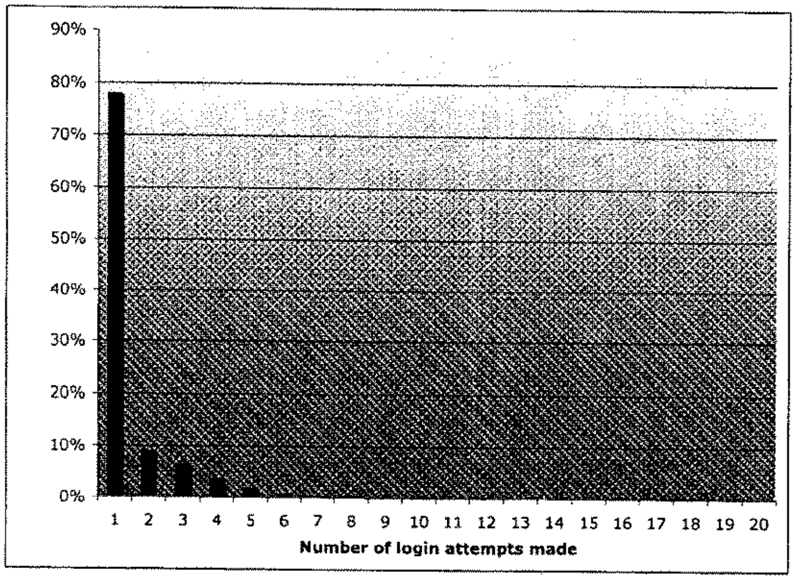


Figure 15: Histogram of occurrences of number of attempts on login for two different hypothetical days (the new one above, and the original below).

The new histogram happens to have more logins that succeed on the first time, and fewer that take a long time – the tail falls off faster (meaning the columns get small quickly as we go to large numbers of login attempts on the x-axis).

However, one of the nice properties of statistics is that if we collect **enough** observations, then the histograms will settle down and converge to the stable value.[9] That will be the underlying probability distribution. We speak of small samples (such as the days above) experiencing *statistical fluctuations* about the true probability distribution. As the amount of data we look at gets large, the fluctuations get smaller (the histograms become more and more like the probability distribution that we would get if we had an extremely large sample).

Let us know look at how we might detect attackers conducting brute force password guessing by looking for statistical anomalies. The simplest scheme we will investigate is called *fixed thresholding*. Here, we simply declare that more than a certain number of login attempts will be an anomaly. Less than that, and we will ignore the problem, more and we will declare there is a problem. Three possible choices of threshold are shown below.

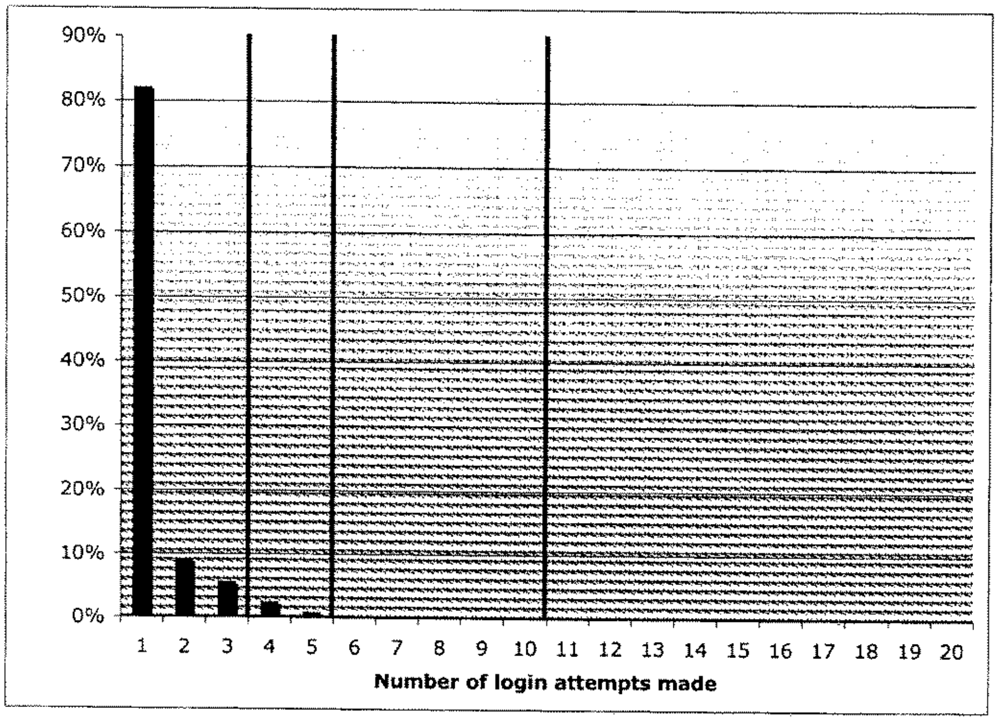


Figure 16: Probability distribution of login attempts, together with three candidate fixed thresholds (at 3, 5, and 10).

[9] This sentence makes the assumption that the data being produced is *stationary* – the distribution of probabilities does not change over time. We will discuss non-stationary situations in a moment.

If we chose the red threshold, then anything more than four login attempts would be considered an anomaly. If we used the blue threshold, then anything more than 5 login attempts would be an anomaly. The green threshold would declare anything over 10 attempts as an anomaly.

Clearly, there is a tradeoff with these thresholds. If we make a low threshold (such as the red one), then on some occasions normal users making innocent mistakes will be viewed as a suspicious anomaly. In particular, if we look at the total odds of a login involving more than 3 attempts in the example above, which we obtain by adding up the size of all columns from 4 out to the right, we find those odds are 3.5%. So in a busy enterprise application with 1000 sessions per day, we would have 35 false alarms per day – enough that busy IT staff would certainly learn to ignore those particular alarms. The proportion of innocent events that will be mistakenly considered to be a problem is known as the *false positive rate*. If we set the threshold to more than 3 as above, we will have a false positive rate of 3.5%. Generally practical systems need to have very low false positive rates, given that the underlying number of events is very large. More than occasional false positives will tend to increase the risk that the system will be ignored. As a commercial practical matter, it is quite hard to achieve satisfactory false positive rates, and the entire industry has struggled with the problem.

On the other hand, if we set the threshold very high, we give would-be attackers more scope than we might wish to guess the password without being detected. For good passwords, brute force will require enormous numbers of guesses that will be easily detectable. However, a certain fraction of users will choose weak guessable passwords given any opportunity to do so, so there is value in minimizing the number of free tries an attacker gets. Situations where the attacker can successfully attack without being detected are known as *false negatives*, and the proportion of attacks that go undetected is known as the *false negative rate.*

The advantage of a threshold scheme is it's simplicity, and in fact simple threshold schemes are still the most widespread in commercial practice. The meaning of a threshold is readily understood by both IT staff and users. However, more complex schemes have been developed, and to those we now turn as we begin to discuss statistical profiles.

The "number of login attempts" variable that we have been studying is an example of what is known in anomaly detection as a *measure.* Statistical anomaly systems frequently have large numbers of measures designed to try to capture different possible things that could go wrong with the system. Some statistical algorithm is applied to each of the measures. We could look for people trying to login to too many different systems, users who look at too many different files, users who login at unusual times, users who come from strange places, etc, etc.

A drawback of threshold schemes is that they expose this complexity (large numbers of measures) to the user of the system. While any individual threshold is easy to understand, if the system has many thresholds understanding them well enough to adjust them may be a prohibitive task. This is particularly true in situations in which the proper value of the threshold will vary from one site to another, and has motivated the search for techniques that will adaptively learn, rather than rely on the software developer to guess the right threshold for the users deployment, and then the user to figure out how to fix it when it is wrong.

Several properties of statistical distributions are important. One is the mean (sometimes informally referred to as the average). The mean is the "center of gravity" of the distribution – if the bars in the histogram were all made of some solid material, the mean is the point on the x-axis where the distribution would teeter at balance, rather than immediately topple to left or right. The mean is the average of all the observations.

Other important concepts are the variance and the standard deviation. The standard deviation is a measure of the average width of a distribution – how far, on average, are the data observations from their mean[10]. Very scattered observations will have a large standard deviation, while tightly clustered ones will have a small standard deviation.

The variance is just the square of the standard deviation.

The standard deviation, often referred to by *sigma* the name of the Greek letter used to denote it, is sometimes used as a different way to assess the anomalousness of an observation. The more "sigmas" an observation is from the mean, the more significant it is taken to be. For example, "two sigmas" is the bare minimum to be noteworthy at all, but three or four sigmas are much better evidence of an anomaly.

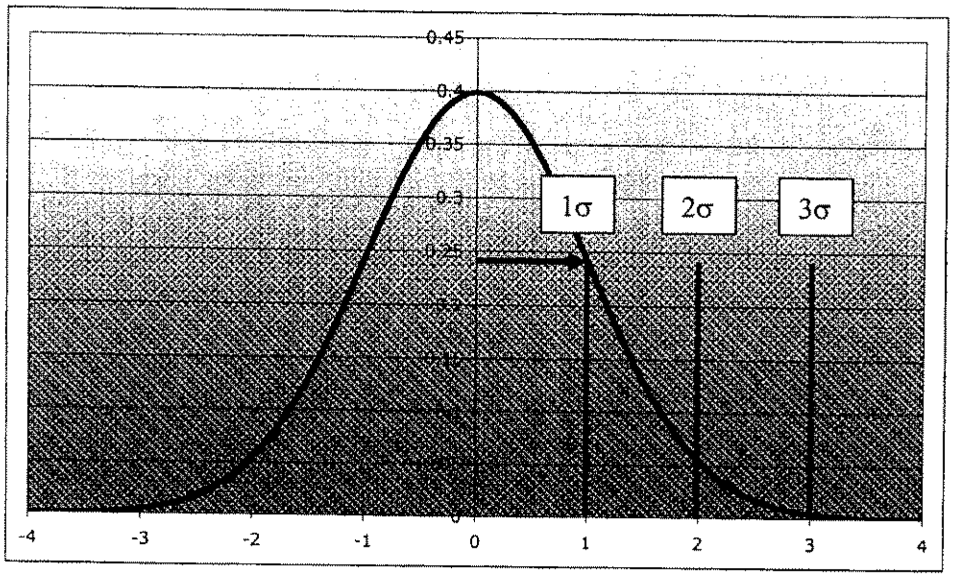


Figure 17: Standard normal distribution. Length of arrow shows the size of the standard deviation. One, two, and three sigmas from mean are shown as black, plum, and blue

[10] Technically, the standard deviation is the root-mean-square distance of the observations from the mean.

This is illustrated above in the case of the *normal distribution,* a mathematically important probability distribution[11]. The mean is at the center of the peak, and the 1σ, 2σ, and 3σ points are shown. Clearly, quite a lot of the distribution is more than one standard deviation from the mean (about 1/3 of the area under the curve). Only about 5% is more than two standard deviations from the mean, and less than 1% is more than 3σ from the mean of a normal distribution (these percentages are properties of this distribution, and would be different in other distributions). Thus for a normally distributed variable, a three sigma anomaly is quite significant.

The normal distribution is an example of a *parametric distribution.* This term refers to a probability distribution that can be described by a closed mathematical expression which accurately captures the distribution. A large body of mathematical techniques are results are available for performing statistical tests for in *parametric statistics*. In the early years of computer intrusion detection, people tried to apply this theory. In general, the goal of building a *statistical profile* is a definition of measures to be studied and some statistical/probabilistic summary that will allow for an efficient determination of whether new observations are anomalies or not. A simple profile would be just to store the mean and standard deviation of the data and then determine how many deviations from the mean new observations are, setting a threshold on that. It has since been realized that parametric distributions are generally a poor representation of the kinds of measures used in this field, and non-parametric methods have become more common. Specifically, measures in networking and network security are often *heavy-tailed* – they fall away to nothing much more slowly than a normal distribution.

Another approach to statistical profiling would be to store an explicit representation of the probability distribution of all observations seen to date. Given a new observation, this would allow a determination of how probable such an observation was (based on how much of the historical probability distribution of the measure was further out in the tail than the latest observation). This unfortunately has drawbacks also. Specifically, measures in realistic situations often have *non-stationary* properties. What this means is that the probability distribution of observations is not fixed over time, but changes for various reasons. This can cause inferences based on historical measurements to be wildly wrong when the distribution changes, resulting in problems of high false-positives, false-negatives, or both.

Thus, an effective statistical profile should learn the changing distribution that the measure is following. This was the motivation for some of the early research in statistical anomaly detection which we will discuss in a later section.

IIc) Background on Computer Security.

For the first three decades of computing, from the development of the ENIAC computer for use in calculating the path of artillery shells in 1945 to roughly the late 1960s, computers were large specialized installations that ran programs delivered to them by direct physical input (eg by feeding stacks of punched cards to a reader). Such computers

[11] Also known as a *Gaussian distribution* in some literature.

were not connected to any networks, phone lines, or even terminals. As such, security threats to them were very similar to security threats to files or documents; the computer and its sensitive data could only be compromised by obtaining physical access to them. Since there was no specialized threat to computers, there was no specialized field of study called computer security.

This began to change in the late 1960s with the advent of time-sharing computers which allowed multiple users to run programs on the system at the same time (beginning with the Dartmouth Time Sharing System in 1964), with the system switching between users rapidly and presenting the illusion to all of simultaneous access (typically via terminals which might be some distance from the computer itself). Such systems allowed each user to have their own storage files and parts of the computer memory. Thus there was the potential for users to read each others files, write over each others files, or for their programs to interfere with each others operation. Thus the field of computer security was born: mechanisms were needed to ensure that users could not interfere with each other.

The early focus of computer security, starting in the late 1970s and early 1980s, was figuring out what kind of security policies the computer should apply, and then attempting to find ways to ensure that the design and implementation of the computer, especially the operating system, could guarantee that the policies would invariably be followed. Much of this work was inspired by the US Department of Defense (DoD) which had a desire to be able to use computers at multiple classification levels while enforcing a policy that would ensure people and processes without secret clearances couldn't access secret computer data, people and processes with only secret clearances couldn't access top-secret data, and so forth.

Researchers quickly realized that a major problem with any kind of access control policy was ensuring that the system actually behaved according to whatever rules its designers and owners had tried to imbue it with. It turned out that there was a huge problem with subtle flaws in the design or implementation that led to ways to circumvent the protection mechanisms. Hackers and vulnerability researchers developed increasingly (and incredibly) ingenious methods for circumventing the rules. A flaw that has the potential to allow the security rules to be circumvented is known as a *vulnerability*, while a technique for taking advantage of a vulnerability is an *exploit*.

For example, one common class of vulnerabilities is called a *buffer overflow*. In these cases, a program is reading in some input (eg from the keyboard) and placing the characters in a region of memory called a buffer. If the programmer makes a mistake in the way this is done and fails to check how much input is being supplied, the program will work normally under almost all circumstances (meaning the error will probably not be found in normal testing). However, an attacker may be able to feed the program specially crafted overlong input which will be written to areas of memory adjacent to the buffer that were holding other critical data that the program was using to keep track of where it was up to in its operations. By supplying carefully crafted input, the attacker can overwrite this status data and cause the program to begin executing new instructions in the computer memory that were supplied by the attacker. Once this has been achieved, the program is then effectively entirely under the control of the attacker and security has been circumvented.

The realization of this kind of problem led to a huge focus on how to verify the correctness of programs and computers as they were being designed and built. This culminated in the Department of Defense "Orange Book" [DoD85]. In addition to mandating features (such as audit trails) that the computer system must have, the rules also covered a variety of processes for reviewing design and implementation to ensure there were no errors that could lead to security problems. The DoD attempted to impose a requirement that computer systems it purchased must conform to Orange Book standards in an attempt to influence the marketplace to adopt more secure computers.

However, this failed. It turned out to be much more time-consuming and expensive to develop systems in a secure manner, so that insecure computers were faster and better in other ways by the time computers designed according to Orange book standards reached the market. Customers preferred having speed and features to having security. Security problems have continued to dog all major deployed brands of computers ever since.

The problems were illustrated through a series of famous hacking cases in the 1980s and 1990s. One example will suffice for here. A young astronomer, Clifford Stoll, was asked to resolve a 75c accounting error in the computer systems at Lawrence Berkeley Lab (one of the Department of Energy's national laboratories). This led to the realization that a hacker had penetrated the computer system. Stoll became obsessed with tracking him down, and eventually was able to find which telephone line the hacker was using so that Stoll could keep a permanent eye on his activities. The hacker was clearly searching for data containing sensitive terms related to the Strategic Defense Initiative (SDI: an 1980s era missile defense program). Stoll concocted an elaborate scheme to create a fake respository of SDI information to cause the hacker to stay on the system long enough to be traced back to his origin. The intrusions turned out to be coming from a German hacker called Marcus Hess, who was selling the information he obtained to the KGB.

A technique that hackers use that is of particular importance to us here is what is today usually known as a *scan*, though in the mid-1990s, the terms *sweep* or *doorknob rattling* where often used for the same general class of activity. The idea of a scan is that an attacker might not know much about the computers on a given network when he began the project of intruding on that network. Thus his first goal would be to perform reconnaissance, and in particular to establish what computers were on the network, and which ones might be vulnerable to means of exploitation available to him. To this end, he will cause his attacking computer to attempt to contact sequentially a number of possible targets and in some manner probe them. In early examples, when guest accounts with default passwords were common, hackers would scan the machines on the network and in each case they would connect to it via telnet and attempt to login using various popular default login and passwords. Sometimes the scanning is systematic – every IP address on the network will systematically be contacted. In other cases it's random. Typically, a program is used to automate the scanning process.

Another development that galvanized the development of computer security was the Internet worm incident of 1988. In this incident, a young Cornell graduate student named Robert Morris released a *worm* – a self-propagating program that broke into computers via one of several exploits it carried and then ran itself on this computers and repeated the process again.